AV Stumpfl GmbH

Documentation

Avio Script

Author:
David Malzner

Version 1.0

Contents

1 Introduction

1.1 About Avio Scripts

Avio Script is based on the script language Lua. Lua has been developed by *Pontical Catholic University of Rio de Janeiro* since 1993 and is open-source software. Lua is thus available as library for external projects and is used by many companies all over the world for the artificial intelligence of characters in computer games or as extension language for programs.
Due to the low resource requirements, the fact that the language can be learned quite easily and is widely distributed we decided to use Lua as a basis for Avio Script. Avio Script is the seamless integration of the script language Lua in the Avio system.

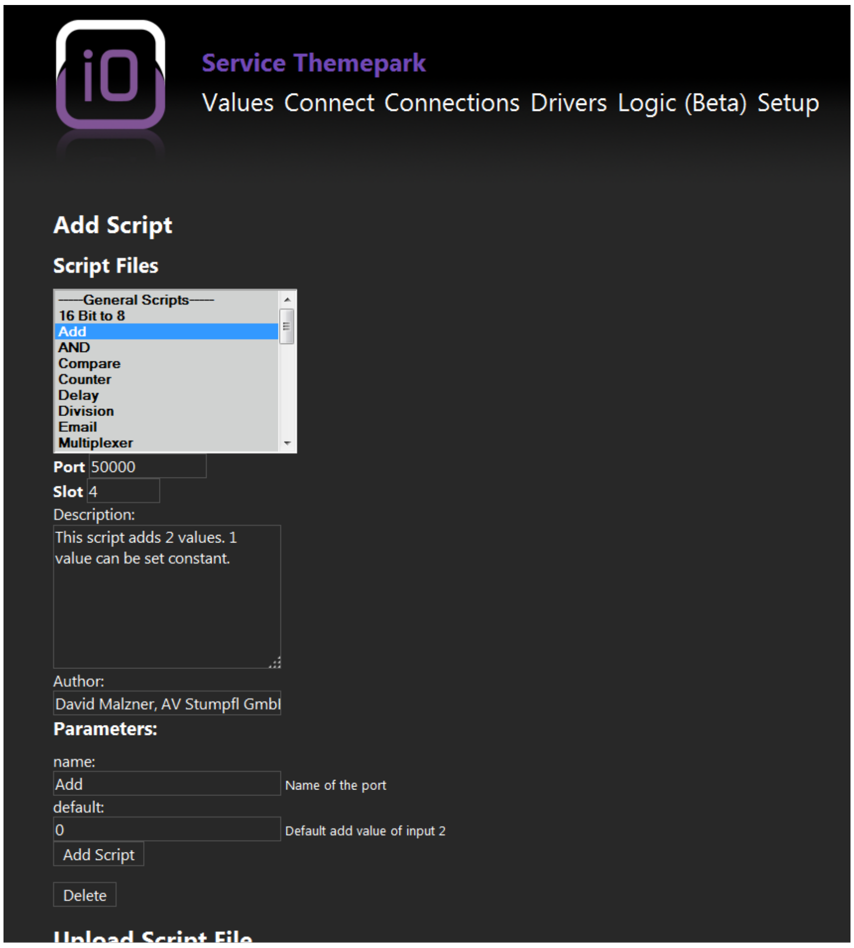
This document is meant to give you a short overview of how to use existing Avio scripts in the Avio system and to serve as an instruction to write your own scripts. Chapter 3 is a quick guide to the script language Lua, giving a short overview of the most important constructs so that you can start programming as fast as possible. This alone should allow you to understand nearly all standard scripts installed along with Avio. For detailed information about Lua take a look at the project website *www.lua.org* which offers a very good documentation.
The target group for this instruction are people who have already been in contact with a programming language although detailed programming skills are not required. It is assumed that the user is familiar with the Avio system.

2 Managing Avio Scripts

During the installation of Avio frequently required scripts are installed as well. These scripts should suffice to meet all the basic requirements of logical operations and are constantly extended as required. The scripts are managed via the web user interface of the corresponding Avio node.

If you open the web user interface of a node all the available scripts are displayed on page *Avio Script*. In section *Add Script* the individual scripts are displayed in a selection box. After selecting the script name, description, author and script parameters to be set are displayed. In the example in Fig. 5 script Add was selected. Possible script parameters are the name of the port used for installing the script and the default value to be added to a value.
For every script to be installed port and slot can be entered additionally. Parameter Port corresponds to the address of the created script port through which the script is later addressed in Avio. This parameter is entered automatically and does not need to be changed unless a particular address should deliberately be used.
Parameter *Slot* refers to the slot used for installing the script. There are 10 slots available and any number of scripts can be installed in one slot. Only 1 script per slot can be run simultaneously . For scripts with mere logical functions without time delay (e.g. *Add*, *And*, *Compare...*) the installed slot is irrelevant. For scripts with a time delay, however, (such as Delay, Ramp,...) any other scripts in the same slot are blocked until the current script has been completed.

Scripts that are in use can be regarded under *Installed Scripts* and the parameters assigned be subsequently changed. Any alterations are immediately recognized by the Avio System and displayed in the Avio Manager (see Fig. 2).
Fig. 3 shows the installed script *Add* in the Avio Manager.



Add Script

Script Files

——General Scripts——
16 Bit to 8
Add
AND
Compare
Counter
Delay
Division
Email
Multiplexer

Port 50000

Slot 4

Description:

This script adds 2 values. 1 value can be set constant.

Author:

David Malzner, AV Stumpfl Gmbl

Parameters:

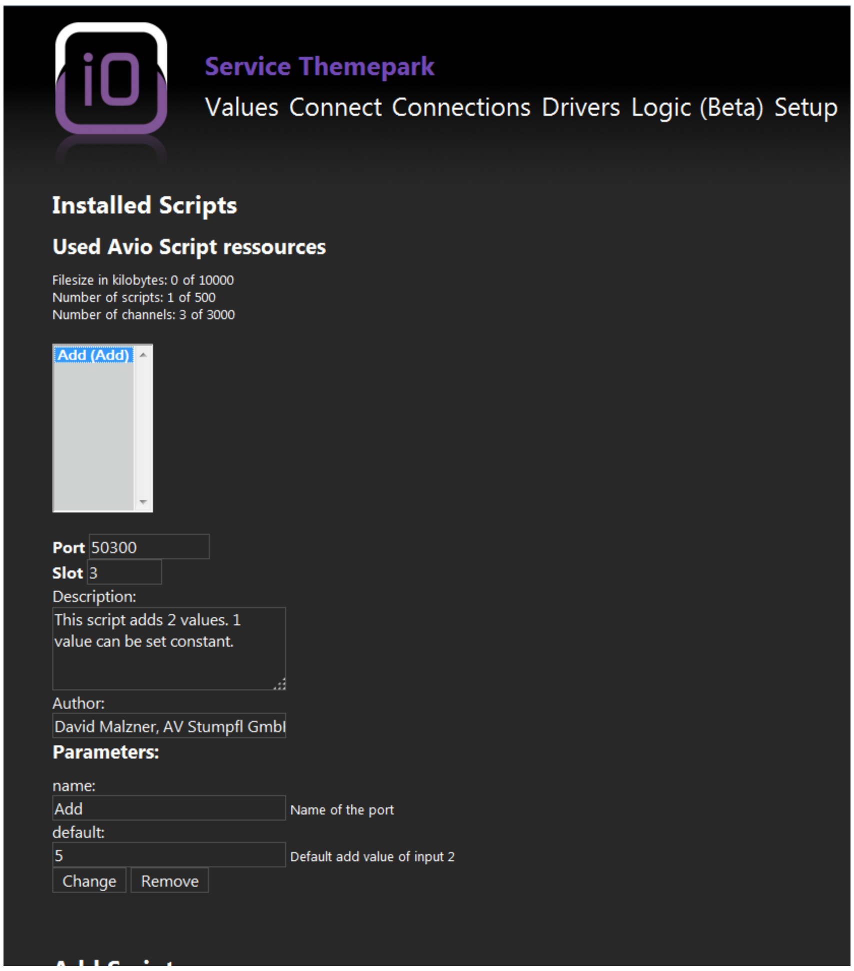
name:

Add — Name of the port

default:

0 — Default add value of input 2

Figure 1: List of available scripts



Installed Scripts

Used Avio Script ressources

Filesize in kilobytes: 0 of 10000
Number of scripts: 1 of 500
Number of channels: 3 of 3000

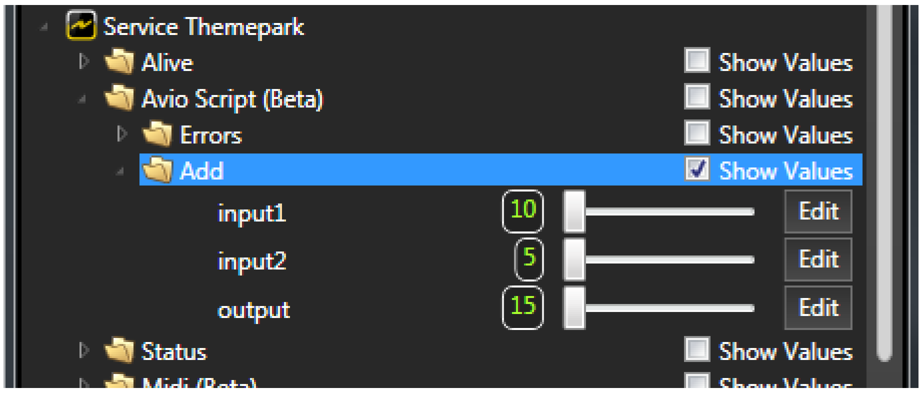
Add (Add)

Port 50300
Slot 3
Description:
This script adds 2 values. 1 value can be set constant.
Author:
David Malzner, AV Stumpfl Gmbl
Parameters:

name:	
Add	Name of the port
default:	
5	Default add value of input 2

Change Remove

Figure 2: List of scripts in use



input1	10	Edit
input2	5	Edit
output	15	Edit

Figure 3: Script Add in the Avio Manager

3 Quick guide to the programming language Lua

This chapters provides a short overview of the programming language Lua. It describes the most important elements necessary to develop your own scripts within a very short time. Although Lua can be learned very easily it also allows quite complex constructs. For a detailed documentation of the programming language refer to the project website *www.lua.org*.
Since Lua is an independent, external programming language the users themselves are responsible for its use, and AV Stumpfl can only provide very limited support. The most important elements, such as functions, variables, conditions and loops can very easily be learned. They alone will help you solve most of the logical operations with Avio Script. Avio Script is based on Lua Version 5.1.
Comments can be commenced using `--` and apply to one line at a time.

3.1 Variables

As is customary for most script languages Lua is a dynamically typed language, i.e. variable types need not be specified before their declaration. This is done during run-time. Before a value is assigned the variable has the value `nil` . The Boolean values `true` and `false` , numbers, strings (delimited by “ “), tables, functions, user data and threads can be assigned to values. In this quick guide we are going to focus on types important for Avio scripts, i.e. Boolean values, numbers, strings and tables. Every variable is globally available even if it was declared within one function. If you want to use the variable only locally you need to use the keyword `local` before assigning a value. Example:

Listing 1: Example of declaration and use of variables

```
x=5;
y="Hello␣from␣Lua";
z=true;
print(x);                          --Output: 5
print(y);                          --Output: "Hello from Lua"
print(z);                          --Output: true
print(a);                          --Output: nil
```

3.2 Tables

Tables are an important and powerful concept of Lua. This quick guide explains how arrays can be used and key-value pairs be saved in it.
An empty table is declared using value { } e.g. `t = {}` . If you want to use the table as an array the individual indexes are selected using a square bracket after the table name, e.g. `t[1] = 3` . Please note that the first element starts with index 1 and not with 0 (it is therefore the *n*th entry that is entered and not the offset relative to the array storage address as some other programming languages do).
It is possible to select arbitrary indexes but it is not necessary to use all indexes. Unused indexes receive value `nil` . Every index may contain any possible data type or one additional table. It is therefore unnecessary that all table entries are the same data type.
The length of a table can be determined using function `table.getn(t)` .
Example:

Listing 2: Example of declaration and usage of tables as array

```
t = {};
t[1] = 5;
```

```
t[2] = 6.5;
t[3] = "Hello␣there";
t[5] = 7;
print(t[1]);                        --Output: 5
print(t[2]);                        --Output: 6.5
print(t[3]);                        --Output: "Hello there"
print(t[4]);                        --Output: nil
print(t[5]);                        --Output: 7
print(table.getn(t))        --Output: 3 (because value 4 is nil, rest of table is not
    counted)
```

An additional way of using tables in Lua is to address values with keys. Instead of the index the name of the key of the table is specified `t["a"] = 5` . Key “a” can also be declared as follows: `t.a = 5` .

Example:

Listing 3: Example of declaration and use of tables as key-value pair

```
t = {};
t["a"] = 5;
t["b"] = 6;
t.c = 7;
print(t["a"]);                  --Output: 5
print(t["b"]);                  --Output: 6
print(t["c"]);                  --Output: 7
```

3.3 Functions

In Lua functions are started with the keyword `function` . This is followed by the name of the function and the input parameters in brackets, e.g. `function foo(input1,input2)` . Functions are closed with `end` . Optionally, values can be returned within functions using the keyword `return` .

Example:

Listing 4: Example Function

```
function add(input1,input2)
    res = input1+input2;
    return res;
end
print(add(7,8));                --Output: 15
```

Functions can also be declared using a variable number of parameters. This is done by entering `...` as parameter, e.g. `function foo(...)` . Following this, a for loop (see chapter 3.5) can be iterated with the individual variable parameters (see Listing 5, line 3). Furthermore, several arguments can also be returned as a result by separating the individual values after the *return* keyword using “,”.

Listing 5: Example of function with variable number of parameters and several return values

```
1  function foo(a,...)
2      print(a);
3      for i,v in ipairs(arg) do
4          print(v)
5      end
6      return a,arg[1];
7  end
8  ret1, ret2 = foo(5,10,11,12,13);          --Output: 5 10 11 12 13
9  print(ret1)                                                   --Output: 5
10 print(ret2)                                                   --Output: 10
```

3.4 If-then-else conditions

If-statements evaluate a condition and execute the corresponding *then* part if the condition is true, otherwise the *else* part. The else part is optional. The block of code is closed with *end*. The following comparisons are possible for an *if* statement:

- == equal
- ˜= not equal
- < less than
- <= less than or equal
- > greater than
- >= greater than or equal
- *and* checks whether left *and* right condition are true
- *or* checks whether left *or* right condition is true

Example:

Listing 6: Example if-then-else condition

```lua
a=5;
if a==5 then
    print("condition met")       --will be executed
end

a=6;
if a==7 then
    print("condition 1 met")     --will not be executed
elseif a==8 then
    print("condition 2 met")     --will not be executed
else
    print("no condition met")    --will be executed
end

a=8
if a~=8 then
    print("condition met")       --will not be executed
else
    print("condition not met")   --will be executed
end

a=9
b=10
if a == 9 and b > 9 then
    print("condition met")       --will be executed
end

a=11;
if(a == 15 or a <= 11) then
    print("condition met")       --will be executed
end
```

3.5 Loops

Loops are areas that can be executed several times one after the other, e.g. counting from 1 to 100. Below we are going to cover the loop types *for* and *while*.

3.5.1 for-loop

Lua uses 2 types of *for* loops, i.e. numeric and generic *for*. For numeric loops a block of code is executed n times. The number of loop passes is defined by a control variable which iterates from a start value to an end value with a defined increment (counting up or down).
A numeric *for* loop has the following syntax:

Listing 7: Syntax numeric *for* loop

```lua
for var=startValue,endValue,incValue do
    something
end
```

The start value *startValue* is assigned to the variable *var*. The variable is incremented with the value *incValue* for every loop pass in which *something* is called until the end value *endValue* is reached. The incremental value *incValue* is optional. Unless specified otherwise, a value of 1 is assumed.

Example:

Listing 8: Example numeric *for* loop

```lua
for i=2,6,2 do       --Output: 2 4 6
    print(i)
end

for i=1,5 do         --Output: 1 2 3 4 5
    print(i)
end
```

For generic loops it is not necessary to specify the value range for iteration. It allows them to loop through all the elements returned by an iterator function.

Listing 9: Syntax generic *for* loop

```lua
for i,v in f(x) do
    something
end
```

Variable i receives the value of the nth loop pass. Variable v receives the value of the current element. Function *f(x)* stands for the iterator function returning all the values to be assigned.

Example:

Listing 10: Example generic *for* loop

```lua
t = {4,5,6,7,8}
for i,v in ipairs(t) do      --Output: 4,5,6,7,8
    print(v)
end
```

In Listing 10 the values of table t are output. Function *ipairs* which is already available in Lua serves as iterator function returning all values of table t.

3.5.2 while-loop

A *while* loop keeps executing a block of code as long as a definable condition is met. It is started with the keyword `while`, the iterating block is started with `do` and closed with `end`.

Example:

Listing 11: Example *while* loop

```
a=10
while a>1 do         --Output: 10 5 2.5 1.25
    print(a)
    a = a/2;
end
```

The example in listing 11 shows a *while* loop, which is executed as long as condition *a>1* is true. During every loop pass variable *a* is halved, i.e. the loop is terminated at a value of 0.75.

3.6 Further information

- Complete documentation of script language Lua version 5.1: `http://www.lua.org/manual/5.1/`
- Documentation of library *LuaXML*: `http://viremo.eludi.net/LuaXML/`
- Documentation of library *LuaSocket* : `http://w3.impa.br/~diego/software/luasocket/introduction.html`

4 Creating Avio Scripts

4.1 Lua Development Tools

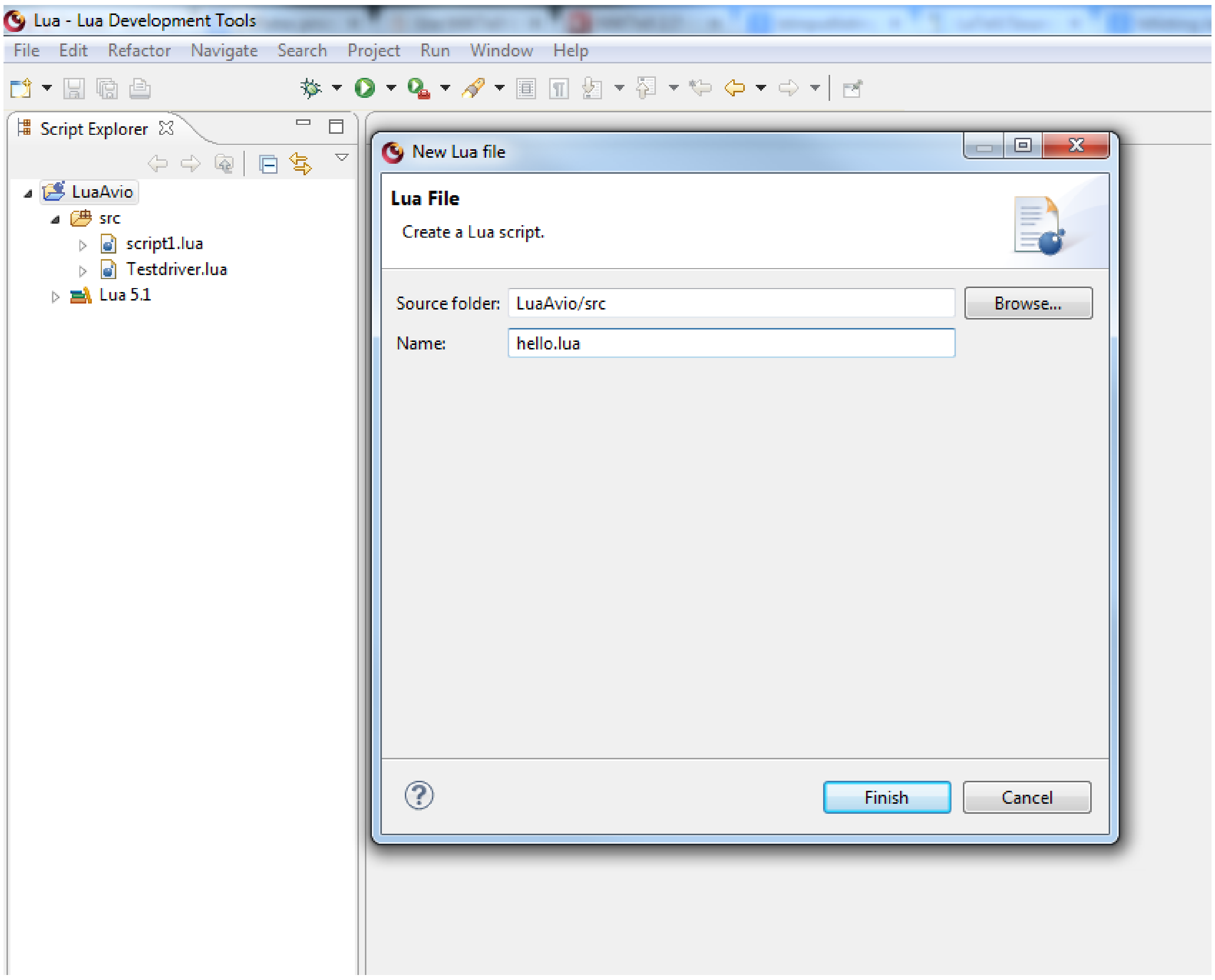
Lua Development Tools [1] is a development environment for the script language Lua. *Lua Development Tools* is based on the development environment Eclipse [2]. *Eclipse* was originally created for the programming language *Java* and is also written in this language. Meanwhile there are various versions and Plugins for many programming languages available. *Eclipse* is open-source software, i.e. the source code is freely available. Furthermore, *Eclipse* was published within the framework of the very liberal *Eclipse Public License* [3] which allows adaptations and distribution thereof.
Lua Development Tools is based on an open-source project by which the development environment *Eclipse* was adapted and extended for the development and debugging of Lua scripts.
Due to the maturity and comfort of this development environment we decided to included it in the delivery as a standard tool for the development and debugging of *Avio Scripts*. This development environment has been integrated into the Avio system such that the scripts created can easily be transferred to the Avio node and debugging of scripts can either be done directly at the Avio node or “offline” in the development environment.

4.2 Creating scripts using "Lua Development Tools"

4.2.1 Adding a new Lua script

After starting *Lua Development Tools* the tree view with the available scripts in the project can be seen on the left side. A file can be added by right-clicking *LuaAvio* and selecting *New - Lua File* in the context menu. In the dialog popping up enter the file name with the file extension *.lua*. This can be seen in Fig. 4. Furthermore, a file can be added to a project by dragging and dropping an existing script into the folder *src* of the tree.

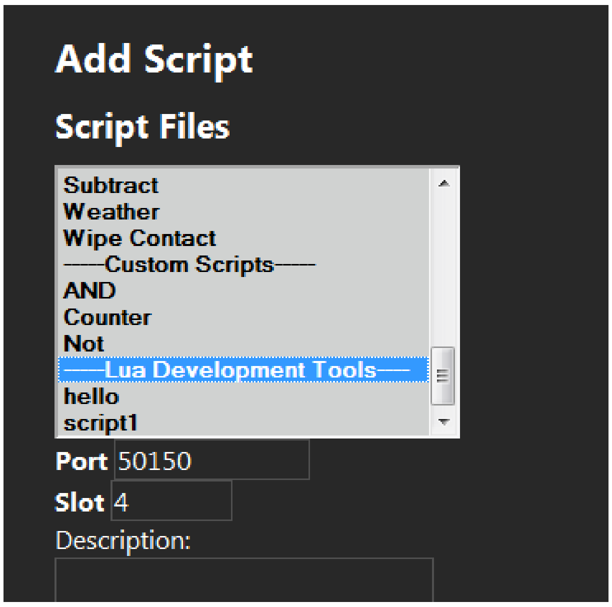
[1]http://www.eclipse.org/koneki/ldt/
[2]http://www.eclipse.org/
[3]http://www.eclipse.org/legal/epl-v10.html



Source folder: LuaAvio/src

Name: hello.lua

Figure 4: Adding a new Lua script



Add Script

Script Files

Subtract
Weather
Wipe Contact
——Custom Scripts——
AND
Counter
Not
——Lua Development Tools——
hello
script1

Port 50150

Slot 4

Description:

Figure 5: Newly added script is available on the web interface

All scripts added to the project are visible on the web interface of the locally installed *Services* just like any other scripts. On page *Avio Script* you can therefore install all scripts of the project in the development environment in section Add Script (see Fig. 5).
If an installed script is modified (no matter in which Editor) an automatic message is sent by Avio Service, pointing out that there had been a modification. These modifications can be accepted with a mouse click and tested in the script.

4.3 Hello World

As is customary for most documentations of programming languages we are also starting with a "Hello World" program, showing the necessary steps to output "Hello World". In the programming language Lua line `print("Hello␣World")` would do. Since we are communicating via the channels of an Avio node in our Avio system the "Hello World" program for Avio is a little more complicated.
The minimal script is displayed in Listing 12.

Listing 12: Hello World

```
1 require("avio")
2 function init()
3     avio.addPort("Hello␣Port","Description␣of␣port.␣This␣is␣displayed␣in␣Wings␣Avio␣
          Manager","string");
4     avio.addChannel("Hello␣Port","Hello␣Channel");
5     avio.setChannel("Hello␣Channel","Hello␣World")
6 end
```

As described in chapter 4.2.1 , any newly created scripts are immediately displayed on the web interface as available scripts. After installing the script a port named "Hello Port" is set up.
Parameter `string` indicates that the data type in the port channel is text. If you want to transfer numbers (data type Integer) just omit this parameter (see Listing 12, line 3).

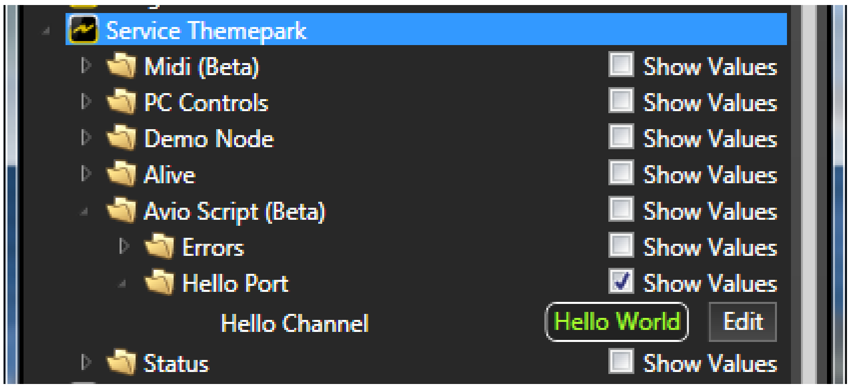


Figure 6: Output of the Hello World script in Avio Manager

In line 4, the channel named "Hello Channel" is added to the port named "Hello Port". In line 5, value "Hello World" is assigned to the channel named "Hello Channel". All this becomes visible in the Avio Manager after installing the script on the web interface (see Fig. 6).

4.4 Avio Library

In addition to the standard libraries, the script language Lua can also be extended by additional libraries. A library is a summary of functions. The Avio library contains all the functions necessary to interact with the Avio system. The Avio library is integrated via the command `require("avio")` . In the documentation below optional parameters are displayed in square brackets [ ]. Text parameters are specified using the prefix *#string*, numeric parameters using prefix *#number*.

4.4.1 addPort

```
avio.addPort(name,description,[datatype])
```

This command adds a new port to the Avio node. A port is a group of channels. Every Avio script requires at least one port to which the channels can later be added and which are available externally.

Parameters:

- *#string name*: port name
- *#string description*: description of the port function This description is displayed in the *Avio Manager*.
- *#string datatype (optional)*: this parameter specifies the data type of the channels that are later added to the port. If value *string* is passed, all channels contained are text channels used for reading and writing character chains. If this parameter is omitted all channels contained allow reading and writing of numeric values.

Return values: none

Example:

```
avio.addPort("Add","This␣port␣adds␣2␣values␣and␣stores␣the␣result␣in␣the␣output␣channel")
```

4.4.2 addChannel

```
avio.addChannel(port,channel,[maxValue])
```

This command adds a channel to a previously created port.

Parameters:

- *#string port*: the name of the port the channel is to be added to. It is only possible to specify ports created in this script.
- *#string channel*: the name of the new channel
- *#number maxValue (optional)*: specifies the maximum value of the channel. This is essential to allow automatic scaling in Avio when connecting channels with differing value ranges. If this parameter is omitted, the maximum value is the maximum value of a signed 32-bit integer ($2^{31} - 1$ or 2.147.483.648). This is also the highest signed number available in Avio.

Return values: none

Example:

```
avio.addChannel("Add","input1")
```

4.4.3 setFunction

```
avio.setFunction(functionName,channel,[channel1],[...])
```

This command defines a function that is called whenever the value of a channel changes. The passed function must contain as many call parameters as channels are passed that call the function after being changed.
If an assigned channel changes while the function is running and blocking, the new function call will be saved as pending call and executed after function finishes. If it is not wanted that pending values will be stored while function is executed, this can be avoided by using the optional flag "noPendingValues".

Parameter:

- *#string function*: The name of the function to be called if one of the subsequently passed channels changes.
- *#string channel1...channelN*: arbitrary number of channels which, after a change in their value, are to call the previous function.
- *#string flags (optional)*: Use flag "noPendingValues" to avoid storing pending function calls while function is executed.

Return values: none

Example:

Listing 13: Example SetFunction

```
avio.setFunction("addFunc","input1","input2")
function addFunc(input1,input2)
    res = input1+input2;
end
```

Listing 14: Example SetFunction without pending values

```
avio.setFunction("delay","start","noPendingValues")
function delay(input)
    avio.sleep(1000);                                     --no pending values will be saved
         while sleeping at this line.
end
```

4.4.4 setChannel

`avio.setChannel(channel,value)`

This command assigns a value to a channel. Only channels that are created in the same script can be used. If the same name was also used in any other script, the channel from the same script is used.

Parameters:

- *#string channel*: name of the channel whose value is to be set
- *#string, #number channel*: the value that is to be passed to the channel.

Return values: none

Example:

```
avio.setChannel("output",5)
avio.setChannel("output","hello␣world")
```

4.4.5 getChannel

`res = avio.getChannel(channel)`

This command reads out the value of a channel. Here, too, and in line with chapter 4.4.4, the name of a channel is used that was created in the same script.

Parameter:

- *#string channel*: name of the channel whose value is to be read

Return values:

- *#number res*: value of the read channel

Example:

```
res = avio.getChannel("input1")
```

4.4.6 sleep

```
avio.sleep(time)
```

This command pauses the executed script for a certain period of time. During this time no functions can be triggered by changing values of a channel (see chapter 4.4.3) in the same script or in any other script located in the same slot.

Parameters:

- *#number time*: time in milliseconds used for pausing

Return values: none

Example:
```
avio.sleep(2000)--Pauses execution of script for 2000 ms (=2 s)
```

4.4.7 setPeriodicFunction

```
avio.setPeriodicFunction(functionname,time)
```

This command defines an existing function to be called periodically in a specific interval.

Parameter:

- *#string function*: Name of function to be called periodically.
- *#number time*: Time in milliseconds between each periodic call of function.

Return values:

- –

Example:
```
avio.setPeriodicFunction("calc",100);
```

4.5 Skeleton of an Avio Script

Every Avio script requires a function named *init*. This function is called by the corresponding Avio node when the script is started. This function initializes the script and creates ports and channels used for script communication with the Avio system (see Hello World Listing 4.3 as a minimal example).
As described in chapter 2 individual scripts can be parameterized and the description be displayed on the Avio node web interface. Information about script and parameterization are written at the beginning of the script file. The description is in Xml format and is later interpreted by the corresponding Avio node whenever the script is added or modified. Since this description is no Lua code it is commented with `--` for Lua. The script description is added to tag `<summary>`, the script name to tag `<name>`.
Individual parameters are defined using tag `<param>`. Every parameter tag receives a name as an argument (argument name *name*) and a default value (argument name *default*) for setting the parameter. The Param-tag contains the description of the parameter. It is useful if every script contains at least one parameter named after the port created during initialization.

When calling the Init function the parameters defined in the header are passed. It is essential that the Init function contains the same number of parameters as defined in the *Xml* script description. The parameters are passed to the Init function in the same order as they were defined. The name of the parameter is irrelevant. Therefore, better readable parameter names can be used in the configuration without having to consider the limitations of a script parameter name. Example Description of the parameter name: "Number of Loops", example of the corresponding parameter name of the Init function: "nrOfLoops". Spaces in the variable names are not possible in Lua.

Listing 15 shows the description of script "Add" as an example.

Listing 15: The *XML* description of script Add at the beginning of the script

```
-- <summary>
--    This script adds 2 values. 1 value can be set constant.
-- </summary>
-- <name>Add</name>
-- <param name="name" default="Add">Name of the port</param>
-- <param name="default" default="0">Default add value of input 2</param>
-- <author>David Malzner, AV Stumpfl GmbH</author>
```

5 Debugging of Avio Scripts in "Lua Development Tools"

In addition to an editor the *Lua Development Tools* also include a powerful debugger. It allows stepping through your own script line by line and monitoring of the variable values. For Avio Scripts two types of debugging are available. There is "offline" debugging on the one hand, which allows the Avio Script only to be tested within the development environment. For more realistic script testing, all Lua libraries, incl. the Avio library (see 4.4) have been added to the Lua Interpreter of the development environment. Since the development environment itself is no Avio node, the script, during "offline" debugging, cannot communicate with the Avio system and is therefore not visible in Avio Manager while it is being executed. If a function from the Avio library is called a corresponding text message appears on the development environment's console, describing what would happen. The advantage of this method is that any modifications can be adopted immediately without having to be transferred to the Avio node and can be tested instantaneously.

5.1 "Offline" without Avio node

5.1.1 Creating the test driver

After starting "Lua Development Tools" two files are already available in the workspace. A file named "script1.lua" showing an example script and a file named "testdriver.lua" which is used for testing other scripts.
In our example we are going to debug the supplied script "Add". Let's drag file *Add.lua* from the script directory *C:\ProgramData\AV Stumpfl\Scripts* and drop it into the workspace (as described in chapter 4.2.1).
Now we want to debug the script. We are going to call the same functions as Avio does in the script. These function calls will be created in file "testdriver.lua".
The chosen file name is only an example, you could choose any name. We want to make file Add.lua visible in the test driver. This is done by line `require("Add")` (see Listing 17, line 2). When loading the script, Avio will call function *init* and the pertaining parameters. In our test

driver this happens in line 3. For parameters name ”Add” and default value ”5”are entered for addition. In the Avio System this can be configured on the web interface (see chapter 2).
The next step is to call function add in the test driver. As defined in listing 16, line 14, this function is called by Avio, if one of the input value changes.

Listing 16: Script add to be tested

```
 1  -- <summary>
 2  --    This script adds 2 values. 1 value can be set constant.
 3  -- </summary>
 4  -- <name>Add</name>
 5  -- <param name="name" default="Add">Name of the port</param>
 6  -- <param name="default" default="0">Default add value of input 2</param>
 7  -- <author>David Malzner, AV Stumpfl GmbH</author>
 8  require("avio")
 9  function init(name,default)
10      avio.addPort(name,"This␣port␣offers␣an␣add␣operation");
11      avio.addChannel(name,"input1");
12      avio.addChannel(name,"input2");
13      avio.addChannel(name,"output");
14      avio.setFunction("add","input1","input2");
15      avio.setChannel("input2",default);
16      print("initialized␣script␣add");
17  end
18
19  function add(input1,input2)
20      res = input1+input2;
21      avio.setChannel("output",res);
22  end
```

Listing 17: Test driver calling the functions to be tested in Script add

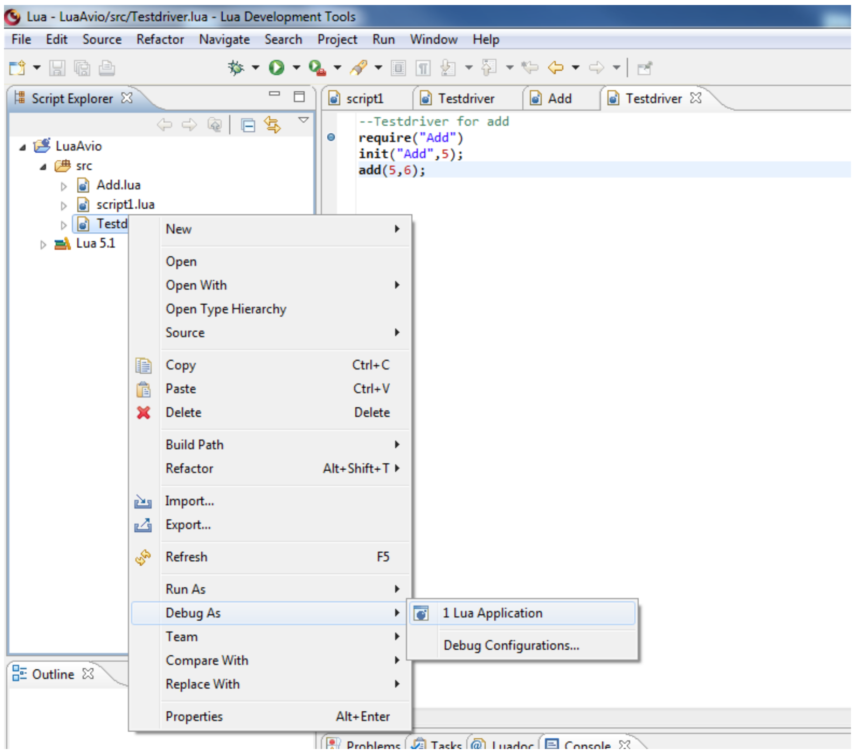
```
1  --Testdriver for add
2  require("Add")
3  init("Add",5);
4  add(5,6);
```

5.1.2 Debugging the test driver

Having created a test driver which calls the script functions in the same way as the Avio System will do later on, we can now debug the script. For the debugger to stop in line 1, we have created a break point in the first line by double-clicking outside the text area. For debugging right-click file ”Testdriver.lua” and select ”Debug As - Lua Application” (see Fig. 7).This is followed by the query whether switching over to the debug perspective of the editor is required. This dialog should be answered in the positive. The development environment is now in debug mode.

Use buttons “Lua” and “Debug” in the top right corner (see Fig. 8) to change over between the perspectives.

Due to the created break point we are now in line 1 of the test driver. Menu entry “Run - Step Over” or pressing the *F6* key executes the current line, “Run - Step Into” or pressing the *F5* key steps into the function of the current line.
The first line containing `require("Add")` serves for making the script to be tested visible (see 5.1.1) and can be stepped over using “Step Over”.
The next to be called is function *init* which is called when the script is loaded and creates the necessary ports and channels (see chapter 4.3). This function can be displayed using “Step Into”. Within this function, you can watch the creation of the individual ports and channels



```
--Testdriver for add
require("Add")
init("Add",5);
add(5,6);
```

Figure 7: Debugging the script with the test driver

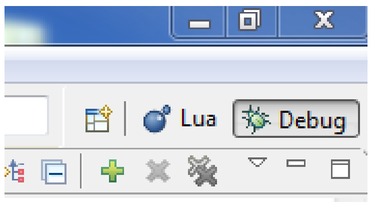
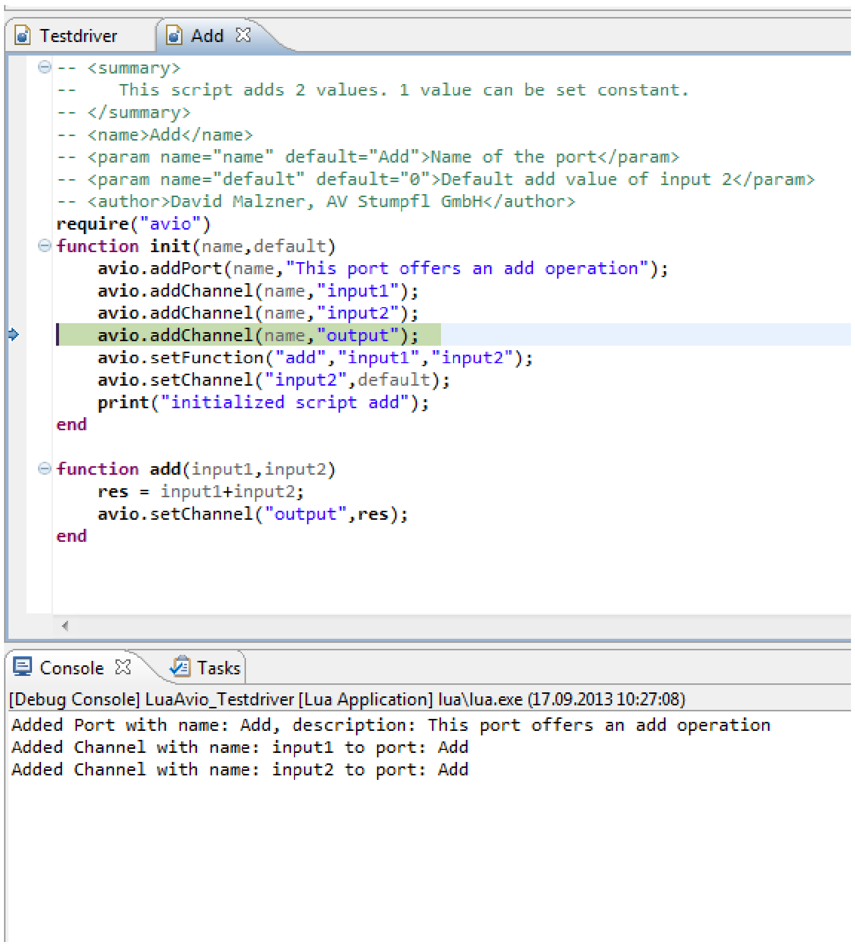


Figure 8: Changing between debugging and development perspective



```
-- <summary>
--    This script adds 2 values. 1 value can be set constant.
-- </summary>
-- <name>Add</name>
-- <param name="name" default="Add">Name of the port</param>
-- <param name="default" default="0">Default add value of input 2</param>
-- <author>David Malzner, AV Stumpfl GmbH</author>
require("avio")
function init(name,default)
    avio.addPort(name,"This port offers an add operation");
    avio.addChannel(name,"input1");
    avio.addChannel(name,"input2");
    avio.addChannel(name,"output");
    avio.setFunction("add","input1","input2");
    avio.setChannel("input2",default);
    print("initialized script add");
end

function add(input1,input2)
    res = input1+input2;
    avio.setChannel("output",res);
end
```

```
[Debug Console] LuaAvio_Testdriver [Lua Application] lua\lua.exe (17.09.2013 10:27:08)
Added Port with name: Add, description: This port offers an add operation
Added Channel with name: input1 to port: Add
Added Channel with name: input2 to port: Add
```

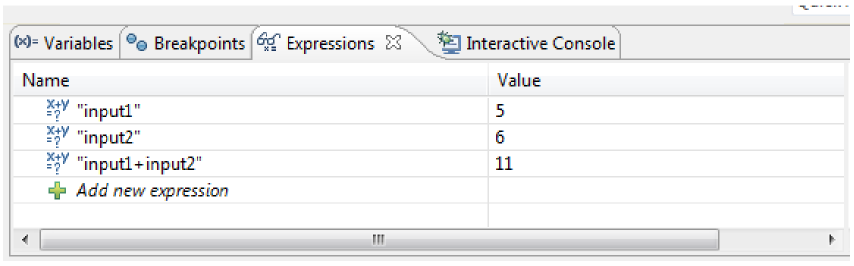
Figure 9: Stepping through the Init function

while stepping through the individual functions via "Step Over". On the console information about the individual functions is displayed in text form (see Fig. 9).

After function *init* was stepped through, use "Step Into" to call function *add* which is later called in Avio if one of the input values to be added changes.
From within the function the variable values can be read out. This can be done by writing the variable name into an empty field in the "Watch Window" on tab "Expressions". Following this the value is displayed next to it (see Fig. 10). It is also possible to evaluate more complex expressions. In our case the two input values *input1* and *input2* are added.

5.2 "Online" debugging on an Avio node

As opposed to "offline" debugging a script can also be debugged when it is already running on an Avio node. The advantage is that it can be tested under real conditions and that no test driver needs to be written. The disadvantage is that any necessary modifications in the script take a little longer.
In this guide "online" debugging is explained on the basis of the script "Add" in chapter 5.1.
For "online" debugging of the script "Add" it is started on the web interface (see chapter 2).
Right-click the created script port in the Avio Manager and a context menu will appear. Here you can select "Debug Script" (see Fig. 11).



Name	Value
"input1"	5
"input2"	6
"input1+input2"	11
Add new expression	

Figure 10: Displaying variable values

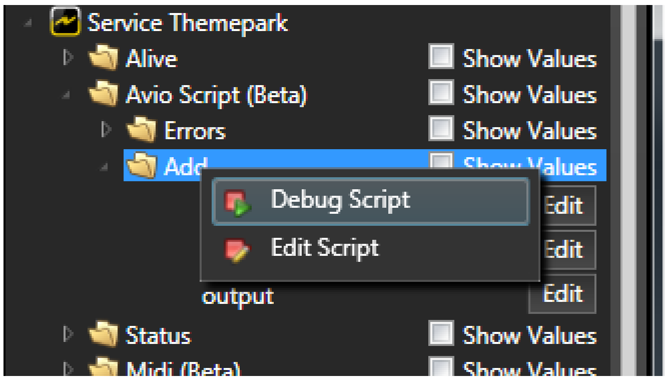
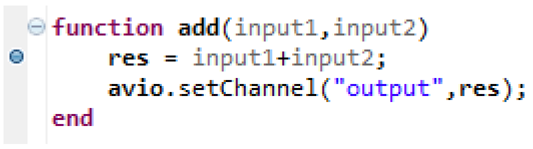


Figure 11: Debugging script in the Manager

The script to be debugged can be located on any node on the network; it need not be started in Avio Service on the local computer. After selecting “Debug Script” a new instance of *Lua Development Tools* with the corresponding settings opens for ”online” debugging. Please note that all running instances of the development environment will be closed at the same time and modifications are not saved.

After starting the development environment’s debugger you will start in line 1 of the Init function. The function can be stepped through line by line as described in the previous chapter using “Step Into” and “Step Over” or run through using “Run - Resume” or pressing the *F8* key. Just as described for “Offline” debugging, individual variables and expressions can also be monitored in the ”Watch Window”. Since we are also interested in what is happening when the *add* function is called we created a breakpoint by double-clicking outside the text area next to the first line of the function. The breakpoint is marked with a blue dot (see Fig. 12).

This function is called every time an input value changes (see Listing 16, line 14 or chapter 4.4.3). To effect such a call you can now change the value of the script input channel in Avio Manager, for example. In the Debugger the currently executed line changes over to the created



```
function add(input1,input2)
    res = input1+input2;
    avio.setChannel("output",res);
end
```

Figure 12: Creating a breakpoint

breakpoint. Now you can continue monitoring the variables in the "Watch Window" and step through the function line by line.

6 Conclusion

This documentation is meant to explain integration of Avio Scripts and the recommended workflow for their creation and debugging. Furthermore, it also contains a short introduction to the script language Lua. In addition to the basics described, Lua, however, has got much more on offer. The standard libraries *os* for "Operating System" allow access to the file system, starting of programs and access to the current time. Library *string* allows complex manipulations of character strings. Library *math* offers all sorts of mathematical operations.

In addition to the standard libraries, libraries *LuaXML* and Socket are also supplied along with the Avio system as described in chapter 3.6. Library *LuaXML* supports parsing and readout of *XML* files. This library is used for the supplied standard script *Weather* in order to read out the weather information called from the Internet in an *XML* file.

Library *Socket* offers full access to the network via *TCP* or *UDP* protocol. Higher functions supported are *SMTP* protocol for sending e-mails and *HTTP* protocol for loading websites. This library is used for the standard scripts *Email* and *Weather.* Documentation of theses libraries would, however, go beyond of the framework for this document. Please refer to the internet pages of the corresponding project (see chapter 3.6).

This guide wants to show you how easy it is to solve nearly all typical logic operations by us using constructs that can very easily be learned.
For a more in-depth look at the script language Lua we recommend the book *Programming in Lua* by *Roberto Ierusalimschy*, one of the chief developers of this programming language, in addition to the numerous online literature that is also available. We would also like to point out that there is always an option of more in-depth study. However, this would go beyond the basic functions of Avio Script and AV Stumpfl is not able to provide any support for this.